FIT3077 - Sprint 4

CL_Monday06pm_Team003

Alex Ung, Romal Patel, Hirun Hettigoda

Table of Contents

Updated UML Diagram

Software Based Prototype

Assumptions:

- When a player swaps with another token whilst in their cave, the swapped player will be positioned on the board outside of the cave. As there is no mention in the brief as to what is required.

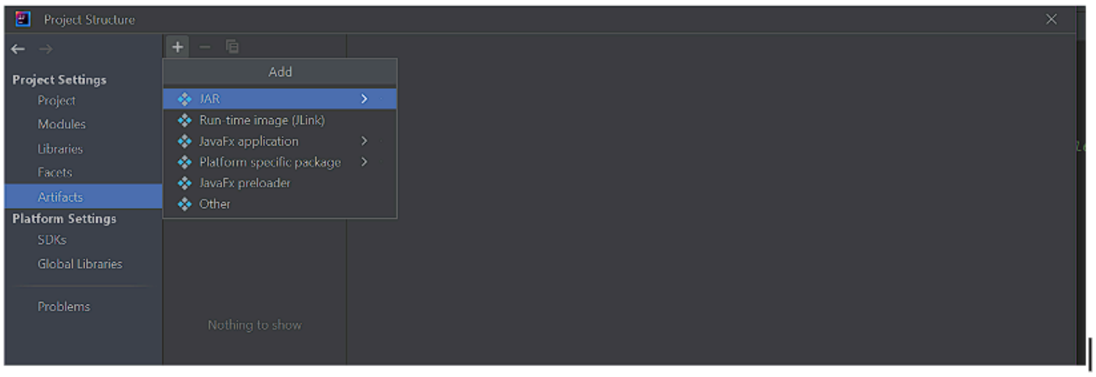
What's required to Run the executable

The user must have jdk 22 installed which can be installed here:
https://www.oracle.com/au/java/technologies/downloads/#jdk22-windows

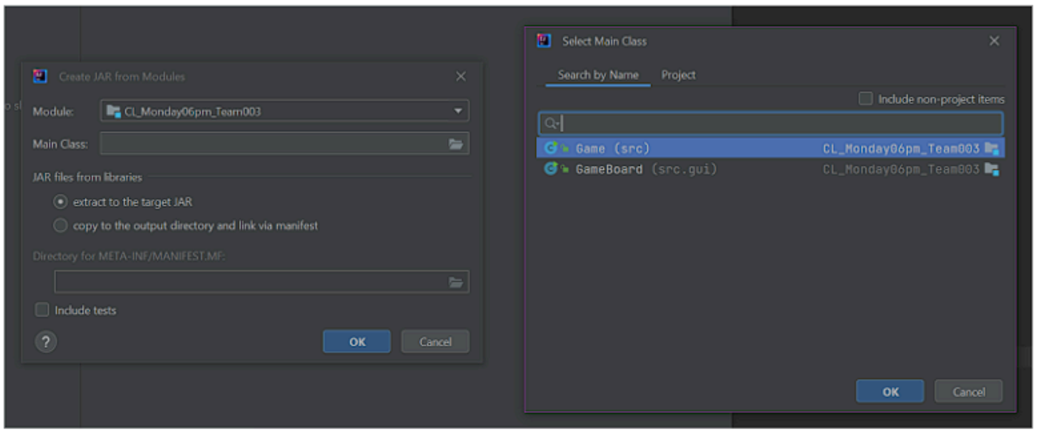
The .jar file can be run on any platform as long as jdk22 is installed.

Instructions for how to build and run the executable

In IntelliJ
Navigate to File -> Project Structure -> Artifacts
Add a new artifact (JAR) (using the + button)



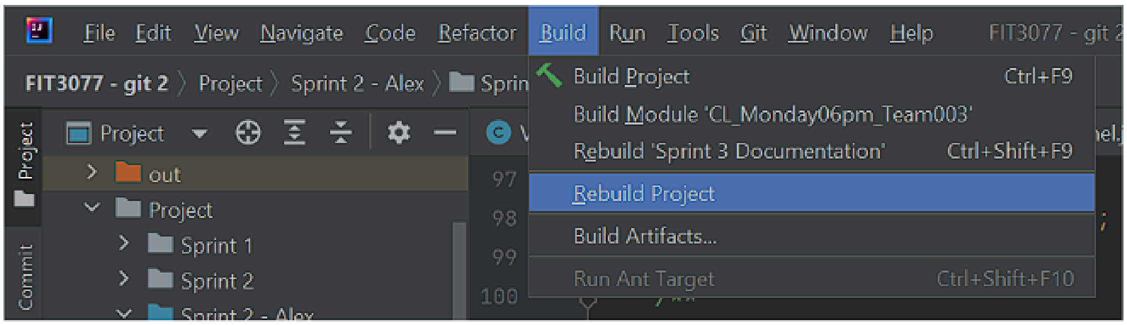
Using the git root folder as the module and import the main class which is the "Game" class


Create JAR from Modules
JAR files from libraries
extract to the target JAR
copy to the output directory and link via manifest
Directory for META-INF/MANIFEST.MF:
Include tests
OK
Cancel
Select Main Class
Search by Name
Project
Include non-project items
OK
Cancel

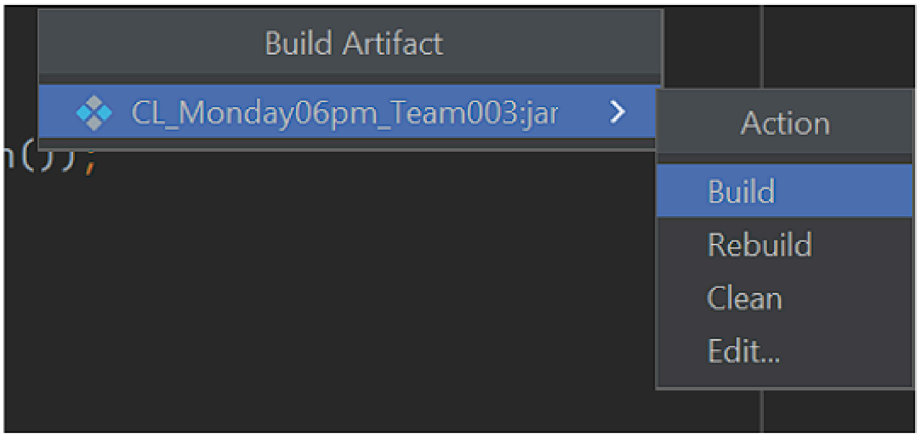
Module: CL_Monday06pm_Team003
Main Class:

Game (src) CL_Monday06pm_Team003
GameBoard (src.gui) CL_Monday06pm_Team003

Specify a location for the output of the .jar file.


File Edit View Navigate Code Refactor Build Run Tools Git Window Help FIT3077 - git 2
FIT3077 - git 2 〉 Project 〉 Sprint 2 - Alex 〉 Sprin
Project
out
Project
Sprint 1
Sprint 2
Sprint 2 - Alex
Build Project Ctrl+F9
Build Module 'CL_Monday06pm_Team003'
Rebuild 'Sprint 3 Documentation' Ctrl+Shift+F9
Rebuild Project
Build Artifacts...
Run Ant Target Ctrl+Shift+F10

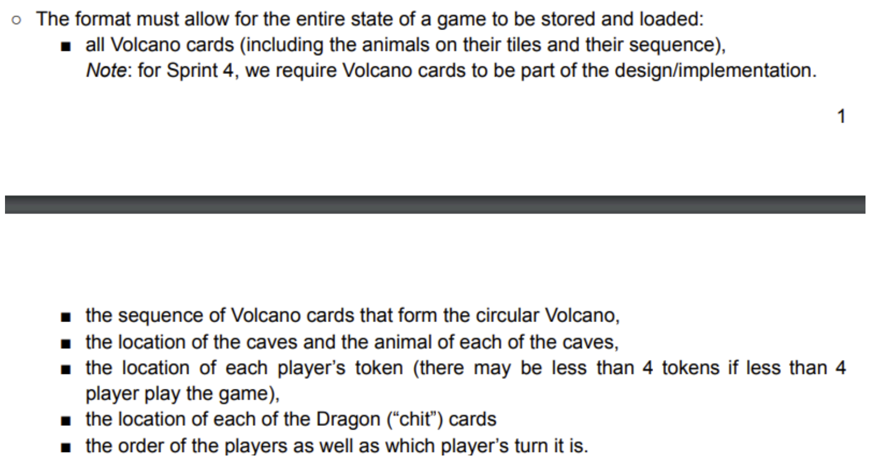
Navigate to build tab and select Build Artifact



Reflection on Sprint 3 Design

Required Extension:

Loading and saving the game from/to an external (configuration) file in a suitable text (not binary) format:



- The format must allow for the entire state of a game to be stored and loaded:
  - all Volcano cards (including the animals on their tiles and their sequence),
    Note: for Sprint 4, we require Volcano cards to be part of the design/implementation.

1

  - the sequence of Volcano cards that form the circular Volcano,
  - the location of the caves and the animal of each of the caves,
  - the location of each player's token (there may be less than 4 tokens if less than 4 player play the game),
  - the location of each of the Dragon ("chit") cards
  - the order of the players as well as which player's turn it is.

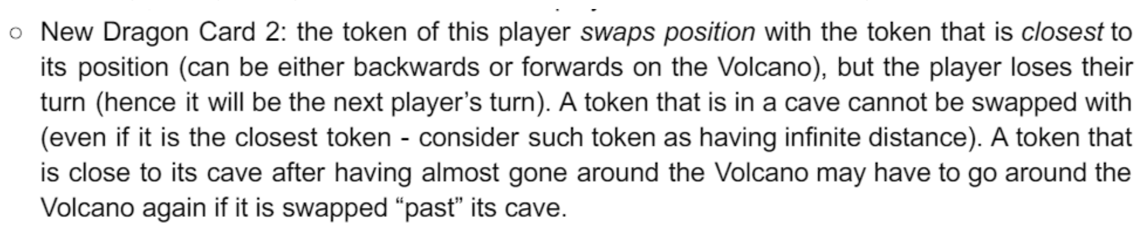
This extension was new in all aspects of the game therefore it required new classes, methods and attributes. To save and load within a game getting and setting different attributes are required, therefore making it simple to get these attributes is the main thing. Since our classes were split up well with the majority of our classes having a single instance following the Singleton design pattern it ensured accuracy and reliability within the game. In the beginning there was a lot to be done in wrapping my head around the whole saving and loading methods however it was well understood especially with the way our game was. This functionality ensures that the game can be saved within the game using a button where the dragon cards, player details, volcano cards, caves and positioning is upheld. This can later be loaded when starting up the game again. I realised that there are a few ways to save and load up a game as mentioned in the draft however as mentioned a text file was the best option since it ensures that the file can be edited.

A difficult thing about incorporating the code from previous sprints into this one was confusion due to lack of consistency. For example in the majority of the classes the images are being added within a separate panel class relating to a key component of that game, like CavePanel for the Cave class. However in some cases images were being added within the WindowPanel class which made it hard to see as there is a large portion of code within that single class. This is another thing which could be changed. There were also a lot of methods present which had similar meaning to other methods making it hard to know which one to use sometimes resulting in a repetition of additional methods. Therefore if we could change anything it would just be to ensure consistency and clear methods to make it easier.

The saving and loading function also required some type of way to load when starting up a game as well as an option to begin the game therefore a menu panel was implemented. This gives the user the option to either start a new game or load up a saved version of the game. This involved moving a bit of code around since there wasn't a menu panel originally, however it was discussed lightly in earlier sprints and if implemented earlier it would've saved much time.

Despite this it was a very good learning experience as saving and loading is important in many factors and our code did much better than expected for making it a better experience.

This Extension has Assumptions.
Implementing a NinjaDragon which performs the effect of the following:



- New Dragon Card 2: the token of this player *swaps position* with the token that is *closest* to its position (can be either backwards or forwards on the Volcano), but the player loses their turn (hence it will be the next player's turn). A token that is in a cave cannot be swapped with (even if it is the closest token - consider such token as having infinite distance). A token that is close to its cave after having almost gone around the Volcano may have to go around the Volcano again if it is swapped "past" its cave.

Ease of implementation:
Overall, the extension was not very difficult to implement. To implement this new dragon card, a new jPanel needed to be added to the dragonCardPool JPanel. The most difficult part was figuring out how to incorporate the ability for a dragonCard to perform a special action that is not constant like the actions of the existing dragonCard creatures. In this particular scenario, a new interface was created for these specialCreatures that perform special actions. These special creatures implement the required method in the interface. This allows for special creatures to be addressed separately from the other creatures.

However, although the extension was easy to implement, there is a presence of not so good oop practices. In this particular case, downcasting had to be used to get access to the special action of the NinjaDragon dragonCard. As a result, this is not in line with the LSP principle where the objects of the superclass should be replaceable of its subclasses without affecting the correctness of the program. In addition, the open/closed principle was violated by having to add the need to modify the existing code to account for "specialCreatures" which could be avoided.

What would the team do differently:
Incorporate polymorphism. Alter the creature class to have an abstract method "perform action" that each creature can implement. This will make things easier when calling the action of the dragon card because the action will be taken care of by the creature itself. In order to not have to repeat overlapping actions, the movement manager already has a method move which can be called pretty easily. This will also remove the need for identifying whether a dragon card is a good, bad or special which will make applying actions easier when flipping a dragon card. This would also make the code follow the open/closed principle as the existing code wouldn't have needed to be changed as adding another dragon card with its own action would be all that is required.

Self-Defined Extension:

The self-defined extension that we chose to go with is to add a card to the board called "Beaver Wizard" that is essentially a chance card each player can use only once per game. When the Beaver Wizard is played (clicked on), the player will move forwards a random amount of spots between 1 and 5. This extension addresses the human value of stimulation where players take on a daring choice that excites them as well as gives them a new and different feel to the game mechanics. It also gives them a chance to be creative with how the game is played and also a sense of curiosity to how the next play will turn out. This targets the self direction human value

In terms of implementing the extension, it was not all that difficult as our design structure was developed in a way to easily introduce new features to the board. In order to implement the Beaver Wizard card, a new JPanel class for the card needed to be created and added to the board which was a fairly simple process given our WindowPanel class handled all entities on the board. Furthermore, the choice to have a singleton class for Player Manager added simplicity in being able to universally access the player tokens. This meant that each action card, including Beaver Wizard, could control the token directly and affect changes on the board without the need for passing multiple references or creating complicated interactions between classes. This approach ensured that we maintained the integrity of our games architecture but also ensured that the addition of new action cards could be managed with minimal disruption to our codebase. This extension complimented our aspects of our modular design well. However, there were some minor challenges to implementation that we can improve on.

It seemed that our method for updating the players turn, turned out to be too convoluted (code smell: long method) and only addressed the player changing turns only when they lost, and therefore displaying a popup message that did not address the actions and theme of Beaver Wizard correctly. The 'updateTurn' method should have been a neutral and universal method that can be used holistically rather than targeting a specific use case. However, this was not a big issue as we disassembled the updateTurn method into its core use: to change the player's turn, and decomposed the rest of the functionality to newer and more refined methods that are to be used more specifically.

Making The Board Dynamic

In order to get the best marks possible, the team decided to make the board as dynamic as possible. Due to time constraints, the team has not implemented the ability to have a varying number of tiles per volcano card, however the player can choose a varying number of volcano cards as well as a varying number of squares per volcano card, but each volcano card will have a constant number of squares.

This was pretty standard to implement. There was a lot of refactoring of existing methods which mainly include the methods inside of the boardArray and volcanoCard class in the backend as well as the windowPanel class for the front end because in the initial stages of the project, namely sprint 3, the team had to work with a configuration that was static for the board size as well as creature distribution. Luckily, most of the surrounding classes follow open/closed principle so classes outside of the boardArray, volcanoCard and windowPanel class did not require any changes.

Since the WindowPanel class is in control of showing the board, most of the UI had to be overhauled to take into account dynamic values and due to the nature of the board that was displayed in sprint 3, odd numbered tiles and volcano cards made it impossible to draw a nice square/rectangle. In light of this, the board is now presented in circular form with caves attached to their associated square(albeit a little awkwardly). However with this new form the board can now be displayed with different configurations.

In terms of being able to implement volcano cards with a varying number of squares, this is most definitely possible, but due to time constraints it just wasn't possible. However, If It were to be implemented, having an input from the user to put in the number of tiles for each volcano card and then using the formula to draw the board in a circular manner accordingly would have been sufficient.

If I were to go back to sprint 3, I would have disregarded the need for the static elements of the code and made the board dynamic from the beginning. This would have prevented the need to refactor pre-existing code to take in these dynamic values. In terms of the code itself, the hardest part was coding the formula required to draw the newly developed circular board which isn't directly related to OOP.

Video Instructions:

1. Video Content overview (Romal 30 secs)
2. Demonstrate technology stack (operating system, IDE) used for testing and implementation (Romal 15 secs)
3. How to make an executable (Alex) (30 secs)
4. Required extension (Alex) (1 min 30secs)
5. Self defined extension (Romal) (1 min 30secs)
6. Saving and loading (Hirun) (1 min 45secs)